25 | MySQL编译器（一）：解析一条SQL语句的执行过程

2020-07-29 宫文学

编译原理实战课 进入课程 >

讲述：宫文学
时长 18:24 大小 16.85M

你好，我是宫文学。现在，就到了我们编译之旅的最后一站了，我们一起来探索一下 MySQL 编译器。

数据库系统能够接受 SQL 语句，并返回数据查询的结果，或者对数据库中的数据进行修改，可以说几乎每个程序员都使用过它。

而 MySQL 又是目前使用最广泛的数据库。所以，解析一下 MySQL 编译并执行 SQL 语句的过程，一方面能帮助你加深对数据库领域的编译技术的理解；另一方面，由于 SQL 是一种最成功的 DSL（特定领域语言），所以理解了 MySQL 编译器的内部运作机制，也能加深你对所有使用数据操作类 DSL 的理解，比如文档数据库的查询语言。另外，解读 SQL 与它的运行时的关系，也有助于你在自己的领域成功地使用 DSL 技术。

那么，数据库系统是如何使用编译技术的呢？接下来，我就会花两讲的时间，带你进入到 MySQL 的内部，做一次全面的探秘。

今天这一讲，我先带你了解一下如何跟踪 MySQL 的运行，了解它处理一个 SQL 语句的过程，以及 MySQL 在词法分析和语法分析方面的实现机制。

好，让我们开始吧！

编译并调试 MySQL

按照惯例，你要下载MySQL 的源代码。我下载的是 8.0 版本的分支。

源代码里的主要目录及其作用如下，我们需要分析的代码基本都在 sql 目录下，它包含了编译器和服务端的核心组件。

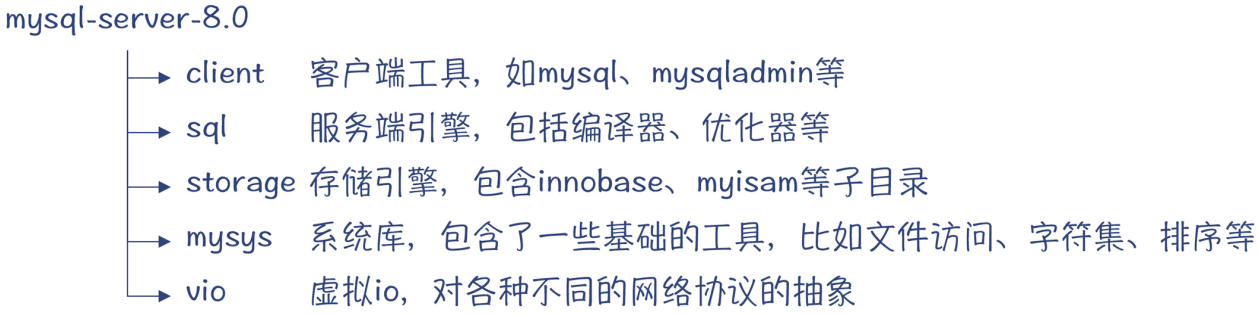


图 1：MySQL 的源代码包含的主要目录

MySQL 的源代码主要是.cc 结尾的，也就是说，MySQL 主要是用 C++ 编写的。另外，也有少量几个代码文件是用 C 语言编写的。

为了跟踪 MySQL 的执行过程，你要用 Debug 模式编译 MySQL，具体步骤可以参考这篇开发者文档。

如果你用单线程编译，大约需要 1 个小时。编译好以后，先初始化出一个数据库来：

复制代码

```
1 ./mysqld --initialize --user=mysql
```

这个过程会为 root@localhost 用户，生成一个缺省的密码。

接着，运行 MySQL 服务器：

```
1 ./mysqld &
```

复制代码

之后，通过客户端连接数据库服务器，这时我们就可以执行 SQL 了：

```
1 ./mysql -uroot -p    #连接mysql server
```

复制代码

最后，我们把 GDB 调试工具附加到 mysqld 进程上，就可以对它进行调试了。

```
1 gdb -p `pidof mysqld`  #pidof是一个工具，用于获取进程的id，你可以安装一下
```

复制代码

提示：这一讲中，我是采用了一个 CentOS 8 的虚拟机来编译和调试 MySQL。我也试过在 macOS 下编译，并用 LLDB 进行调试，也一样方便。

注意，你在调试程序的时候，有两个**设置断点**的好地方：

> **dispatch_command**：在 sql/sql_parse.cc 文件里。在接受客户端请求的时候（比如一个 SQL 语句），会在这里集中处理。
>
> **my_message_sql**：在 sql/mysqld.cc 文件里。当系统需要输出错误信息的时候，会在这里集中处理。

这个时候，我们在 MySQL 的客户端输入一个查询命令，就可以从雇员表里查询姓和名了。在这个例子中，我采用的数据库是 MySQL 的一个示例数据库 employees，你可以根据它的文档来生成示例数据库。

```
1 mysql> select first_name, last_name from employees;    #从mysql库的user表中查询信
```

复制代码

这个命令被 mysqld 接收到以后，就会触发断点，并停止执行。这个时候，客户端也会老老实实地停在那里，等候从服务端传回数据。即使你在后端跟踪代码的过程会花很长的时间，客户端也不会超时，一直在安静地等待。给我的感觉就是，MySQL 对于调试程序还是很友好的。

在 GDB 中输入 bt 命令，会打印出调用栈，这样你就能了解一个 SQL 语句，在 MySQL 中执行的完整过程。为了方便你理解和复习，这里我整理成了一个表格：

程序文件	函数	说明
libsystem_pthread.dylib	thread_start	启动新线程
libsystem_pthread.dylib	_pthread_start	
storage/perfschema/pfs.cc	pfs_spawn_thread()	为新创建的线程做初始化 perfschema是Performance Schema的缩写，其作用是测量服务器的执行性能，并且在运行时提供自省功能
sql/conn_handler/connection_handler_per_thread.cc	handle_connection()	处理一个连接的相关线程 包括： *初始化线程和相关数据结构 *对用户认证 *执行这个连接发送来的所有查询 *结束连接 *结束线程
sql/sql_parse.cc	do_command()	从连接中读取一个命令并执行
sql/sql_parse.cc	dispatch_command()	根据命令的不同种类，调用不同的程序去处理
sql/sql_parse.cc	mysql_parse()	解析一个SQL语句，并执行
sql/sql_parse.cc	parse_sql()	把一个SQL语句翻译成AST
sql/sql_class.cc	sql_parser()	调用解析器获得解析树(parse tree) 再把解析树翻译成AST（调用sql_lex.cc.make_sql_cmd()） sql_class.cc定义了MySQL所用到的一个很基础的类THD和它所依赖的类
sql/sql_yacc.cc	MYSQLparse()	这段源代码是由bison根据sql_yacc.yy生成的

我也把 MySQL 执行 SQL 语句时的一些重要程序入口记录了下来，这也需要你重点关注。它反映了执行 SQL 过程中的一些重要的处理阶段，包括语法分析、处理上下文、引用消解、优化和执行。你在这些地方都可以设置断点。

图 2：MySQL 执行 SQL 语句时的部分重要程序入口

好了，现在你就已经做好准备，能够分析 MySQL 的内部实现机制了。不过，由于 MySQL 执行的是 SQL 语言，它跟我们前面分析的高级语言有所不同。所以，我们先稍微回顾一下 SQL 语言的特点。

SQL 语言：数据库领域的 DSL

SQL 是结构化查询语言（Structural Query Language）的英文缩写。举个例子，这是一个很简单的 SQL 语句：

复制代码

```
1 select emp_no, first_name, last_name from employees;
```

其实在大部分情况下，SQL 都是这样一个一个来做语句执行的。这些语句又分为 DML（数据操纵语言）和 DDL（数据定义语言）两类。前者是对数据的查询、修改和删除等操作，而后者是用来定义数据库和表的结构（又叫模式）。

我们平常最多使用的是 DML。而 DML 中，执行起来最复杂的是 select 语句。所以，在本讲，我都是用 select 语句来给你举例子。

那么，SQL 跟我们前面分析的高级语言相比有什么不同呢？

第一个特点：SQL 是声明式（Declarative）的。这是什么意思呢？其实就是说，SQL 语句能够表达它的计算逻辑，但它不需要描述控制流。

高级语言一般都有控制流，也就是详细规定了实现一个功能的流程：先调用什么功能，再调用什么功能，比如 if 语句、循环语句等等。这种方式叫做**命令式（imperative）编程**。

更深入一点，声明式编程说的是“要什么”，它不关心实现的过程；而命令式编程强调的是“如何做”。前者更接近人类社会的领域问题，而后者更接近计算机实现。

第二个特点：SQL 是一种特定领域语言（DSL，Domain Specific Language），专门针对关系数据库这个领域的。SQL 中的各个元素能够映射成关系代数中的操作术语，比如选择、投影、连接、笛卡尔积、交集、并集等操作。它采用的是表、字段、连接等要素，而不需要使用常见的高级语言的变量、类、函数等要素。

所以，SQL 就给其他 DSL 的设计提供了一个很好的参考：

> **采用声明式，更加贴近领域需求。**比如，你可以设计一个报表的 DSL，这个 DSL 只需要描述报表的特征，而不需要描述其实现过程。
>
> **采用特定领域的模型、术语，甚至是数学理论。**比如，针对人工智能领域，你完全就可以用张量计算（力学概念）的术语来定义 DSL。

好了，现在我们分析了 SQL 的特点，从而也让你了解了 DSL 的一些共性特点。那么接下来，顺着 MySQL 运行的脉络，我们先来了解一下 MySQL 是如何做词法分析和语法分析的。

词法和语法分析

词法分析的代码是在 sql/sql_lex.cc 中，入口是 MYSQLlex() 函数。在 sql/lex.h 中，有一个 symbols[]数组，它定义了各类关键字、操作符。

MySQL 的词法分析器也是手写的，这给算法提供了一定的灵活性。比如，SQL 语句中，Token 的解析是跟当前使用的字符集有关的。使用不同的字符集，词法分析器所占用的字节数是不一样的，判断合法字符的依据也是不同的。而字符集信息，取决于当前的系统的配置。词法分析器可以根据这些配置信息，正确地解析标识符和字符串。

MySQL 的语法分析器是用 bison 工具生成的，bison 是一个语法分析器生成工具，它是 GNU 版本的 yacc。bison 支持的语法分析算法是 LALR 算法，而 LALR 是 LR 算法家族中的一员，它能够支持大部分常见的语法规则。bison 的规则文件是 sql/sql_yacc.yy，经过编译后会生成 sql/sql_yacc.cc 文件。

sql_yacc.yy 中，用你熟悉的 EBNF 格式定义了 MySQL 的语法规则。我节选了与 select 语句有关的规则，如下所示，从中你可以体会一下，SQL 语句的语法是怎样被一层一层定义出来的：

复制代码

```
select_stmt:
          query_expression
        | ...
        | select_stmt_with_into
        ;
query_expression:
          query_expression_body opt_order_clause opt_limit_clause
        | with_clause query_expression_body opt_order_clause opt_limit_clause
        | ...
        ;
query_expression_body:
          query_primary
        | query_expression_body UNION_SYM union_option query_primary
        | ...
        ;
query_primary:
          query_specification
        | table_value_constructor
        | explicit_table
        ;
query_specification:
          ...
```

```
        | SELECT_SYM              /*select关键字*/
          select_options          /*distinct等选项*/
          select_item_list        /*select项列表*/
          opt_from_clause         /*可选：from子句*/
          opt_where_clause        /*可选：where子句*/
          opt_group_clause        /*可选：group子句*/
          opt_having_clause       /*可选：having子句*/
          opt_window_clause       /*可选：window子句*/
        ;

```

其中，query_expression 就是一个最基础的 select 语句，它包含了 SELECT 关键字、字段列表、from 子句、where 子句等。

你可以看一下 select_options、opt_from_clause 和其他几个以 opt 开头的规则，它们都是 SQL 语句的组成部分。opt 是可选的意思，也就是它的产生式可能产生ε。

复制代码

```
opt_from_clause:
          /* Empty. */
        | from_clause
        ;

```

另外，你还可以看一下表达式部分的语法。在 MySQL 编译器当中，对于二元运算，你可以大胆地写成左递归的文法。因为它的语法分析的算法用的是 LALR，这个算法能够自动处理左递归。

一般研究表达式的时候，我们总是会关注编译器是如何处理结合性和优先级的。那么，bison 是如何处理的呢？

原来，bison 里面有专门的规则，可以规定运算符的优先级和结合性。在 sql_yacc.yy 中，你会看到如下所示的规则片段：

```
%left CONDITIONLESS_JOIN
%left   JOIN_SYM INNER_SYM CROSS STRAIGHT_JOIN NATURAL LEFT RIGHT ON_SYM USING
%left   SET_VAR
%left   OR_SYM OR2_SYM
%left   XOR
%left   AND_SYM AND_AND_SYM
%left   BETWEEN_SYM CASE_SYM WHEN_SYM THEN_SYM ELSE
%left   EQ EQUAL_SYM GE GT_SYM LE LT NE IS LIKE REGEXP IN_SYM
%left   '|'
%left   '&'
%left   SHIFT_LEFT SHIFT_RIGHT
%left   '-' '+'
%left   '*' '/' '%' DIV_SYM MOD_SYM
%left   '^'
%left   OR_OR_SYM
%left   NEG '~'
%right  NOT_SYM NOT2_SYM
%right  BINARY_SYM COLLATE_SYM
%left  INTERVAL_SYM
%left SUBQUERY_AS_EXPR
%left '(' ')'
```

你可以看一下 bit_expr 的产生式，它其实完全把加减乘数等运算符并列就行了。

复制代码

```
bit_expr :
          ...
          | bit_expr '+' bit_expr %prec '+'
          | bit_expr '-' bit_expr %prec '-'
          | bit_expr '*' bit_expr %prec '*'
          | bit_expr '/' bit_expr %prec '/'
          ...
          | simple_expr
```

如果你只是用到加减乘除的运算，那就可以不用在产生式的后面加 %prec 这个标记。但由于加减乘除这几个还可以用在其他地方，比如“-a”可以用来表示把 a 取负值；减号可以用在一元表达式当中，这会比用在二元表达式中有更高的优先级。也就是说，为了区分同一个 Token 在不同上下文中的优先级，我们可以用 %prec，来说明该优先级是上下文依赖的。

好了，在了解了词法分析器和语法分析器以后，我们接着来跟踪一下 MySQL 的执行，看看编译器所生成的解析树和 AST 是什么样子的。

在 sql_class.cc 的 sql_parser() 方法中，编译器执行完解析程序之后，会返回解析树的根节点 root，在 GDB 中通过 p 命令，可以逐步打印出整个解析树。你会看到，它的根节点是一个 PT_select_stmt 指针（见图 3）。

解析树的节点是在语法规则中规定的，这是一些 C++ 的代码，它们会嵌入到语法规则中去。

下面展示的这个语法规则就表明，编译器在解析完 query_expression 规则以后，要创建一个 PT_query_expression 的节点，其构造函数的参数分别是三个子规则所形成的节点。对于 query_expression_body 和 query_primary 这两个规则，它们会直接把子节点返回，因为它们都只有一个子节点。这样就会简化解析树，让它更像一棵 AST。关于 AST 和解析树（也叫 CST）的区别，我在解析 Python 的编译器中讲过了，你可以回忆一下。

复制代码

```
query_expression:
          query_expression_body
          opt_order_clause
          opt_limit_clause
          {
            $$ = NEW_PTN PT_query_expression($1, $2, $3); /*创建节点*/
          }
        | ...

query_expression_body:
          query_primary
          {
            $$ = $1;    /*直接返回query_primary的节点*/
          }
        | ...

query_primary:
          query_specification
          {
            $$= $1;   /*直接返回query_specification的节点*/
          }
        | ...

```

最后，对于“select first_name, last_name from employees”这样一个简单的 SQL 语句，它所形成的解析树如下：

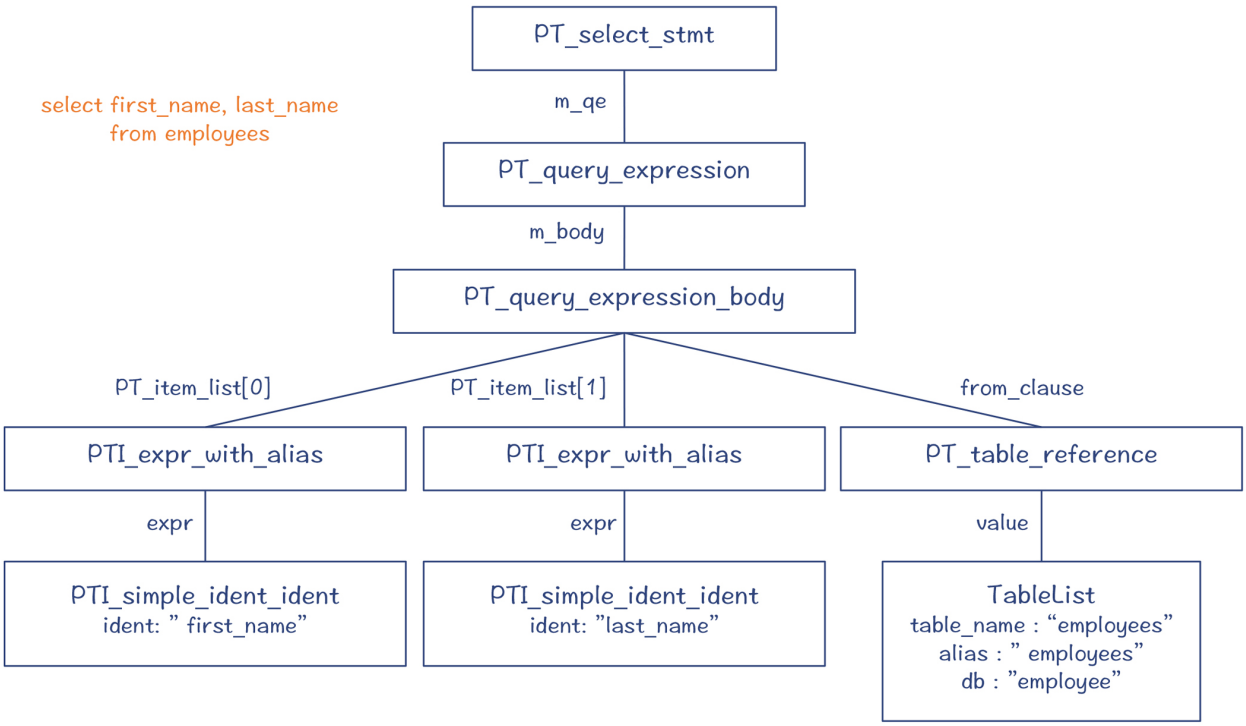


图 3：示例 SQL 解析后生成的解析树

而对于“select 2 + 3”这样一个做表达式计算的 SQL 语句，所形成的解析树如下。你会看到，它跟普通的高级语言的表达式的 AST 很相似：

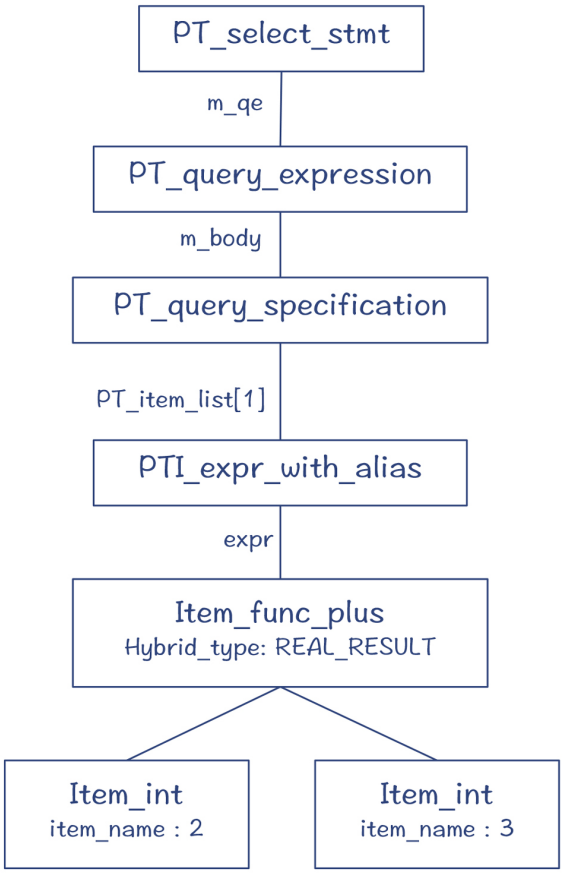


图 4： “select 2 + 3”对应的解析树

图 4 中的 PT_query_expression 等类，就是解析树的节点，它们都是 Parse_tree_node 的子类（PT 是 Parse Tree 的缩写）。这些类主要定义在 sql/parse_tree_nodes.h 和 parse_tree_items.h 文件中。

其中，Item 代表了与“值”有关的节点，它的子类能够用于表示字段、常量和表达式等。你可以通过 Item 的 val_int()、val_str() 等方法获取它的值。

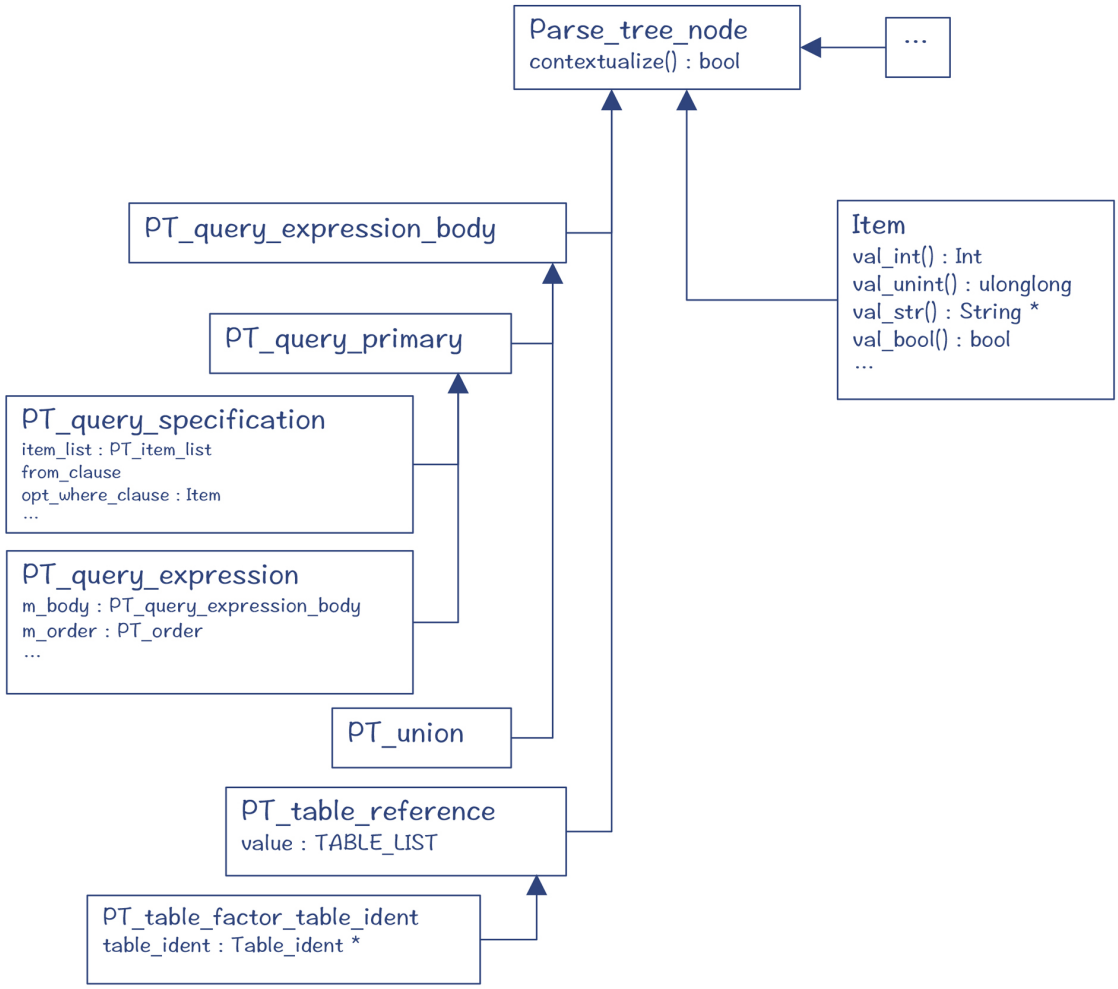


图 5：解析树的树节点（部分）

由于 SQL 是一个个单独的语句，所以 select、insert、update 等语句，它们都各自有不同的根节点，都是 Parse_tree_root 的子类。

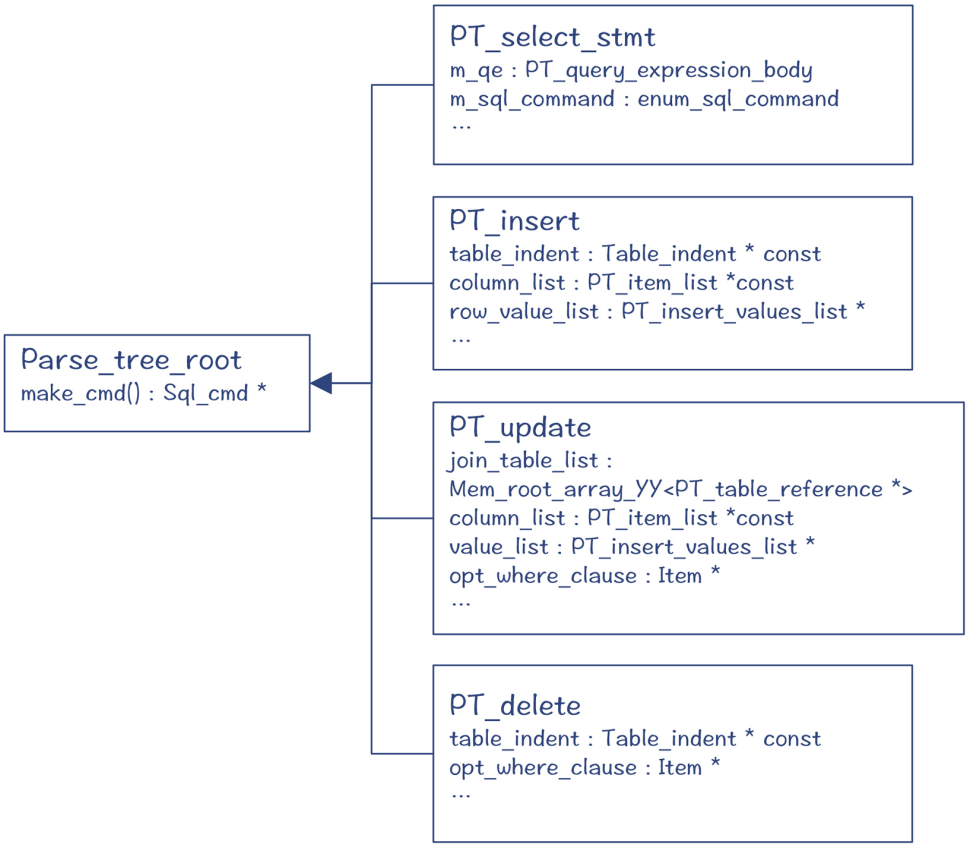


图 6：解析树的根节点

好了，现在你就已经了解了 SQL 的解析过程和它所生成的 AST 了。前面我说过，MySQL 采用的是 LALR 算法，因此我们可以借助 MySQL 编译器，来加深一下对 LR 算法家族的理解。

重温 LR 算法

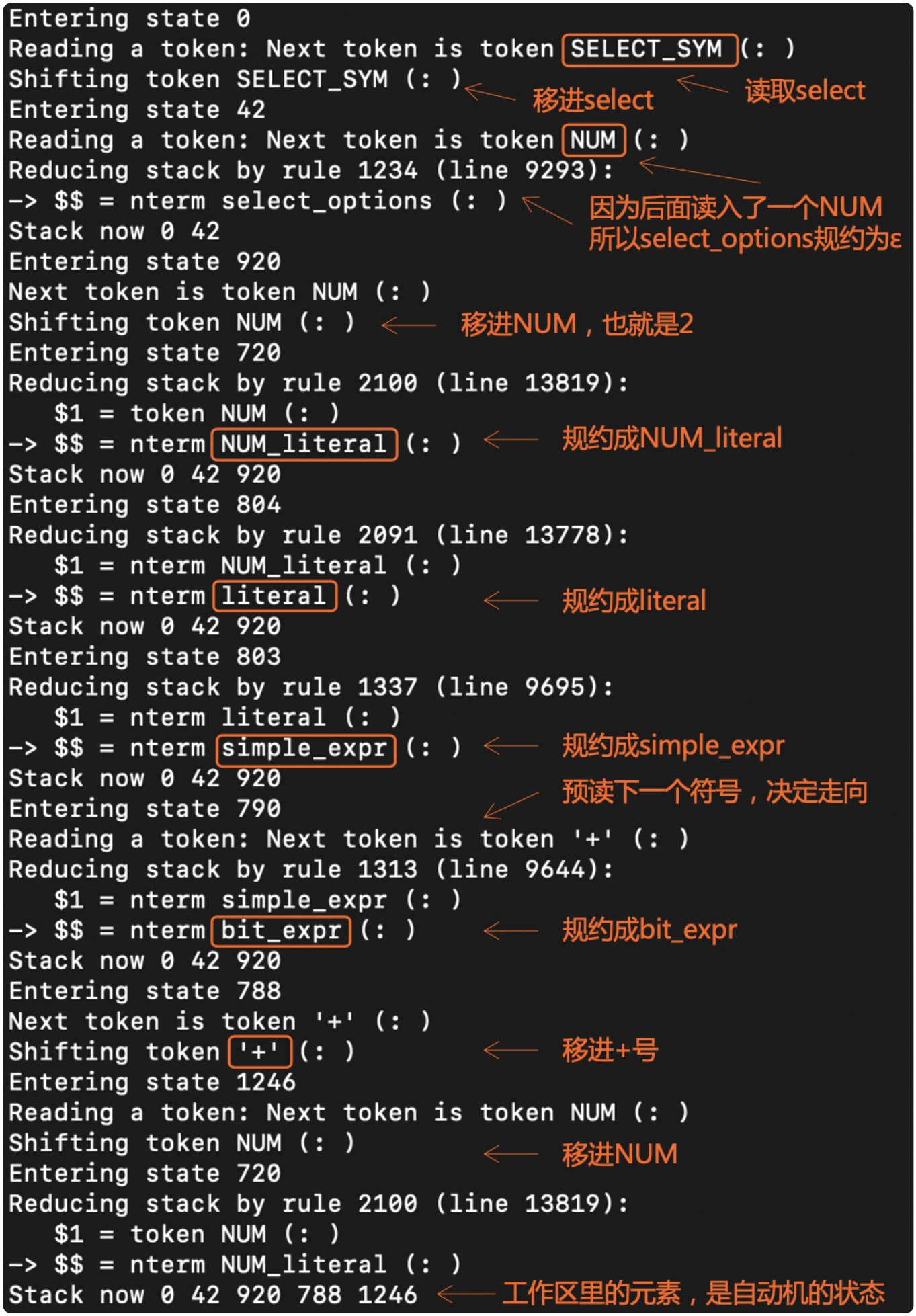
你在阅读 yacc.yy 文件的时候，在注释里，你会发现如何跟踪语法分析器的执行过程的一些信息。

你可以用下面的命令，带上“-debug”参数，来启动 MySQL 服务器：

```
1 mysqld --debug="d,parser_debug"
```

复制代码

然后，你可以通过客户端执行一个简单的 SQL 语句：“select 2+3*5”。在终端，会输出语法分析的过程。这里我截取了一部分界面，通过这些输出信息，你能看出 LR 算法执行过程中的移进、规约过程，以及工作区内和预读的信息。



我来给你简单地复现一下这个解析过程。

第 1 步，编译器处于状态 0，并且预读了一个 select 关键字。你已经知道，LR 算法是基于一个 DFA 的。在这里的输出信息中，你能看到某些状态的编号达到了一千多，所以这个 DFA 还是比较大的。

第 2 步，把 select 关键字移进工作区，并进入状态 42。这个时候，编译器已经知道后面跟着的一定是一个 select 语句了，也就是会使用下面的语法规则：

复制代码

```
query_specification:
          ...
        | SELECT_SYM             /*select关键字*/
          select_options         /*distinct等选项*/
          select_item_list       /*select项列表*/
          opt_from_clause        /*可选：from子句*/
          opt_where_clause       /*可选：where子句*/
          opt_group_clause       /*可选：group子句*/
          opt_having_clause      /*可选：having子句*/
          opt_window_clause      /*可选：window子句*/
        ;

```

为了给你一个直观的印象，这里我画了 DFA 的局部示意图（做了一定的简化），如下所示。你可以看到，在状态 42，点符号位于“select”关键字之后、select_options 之前。select_options 代表了“distinct”这样的一些关键字，但也有可能为空。

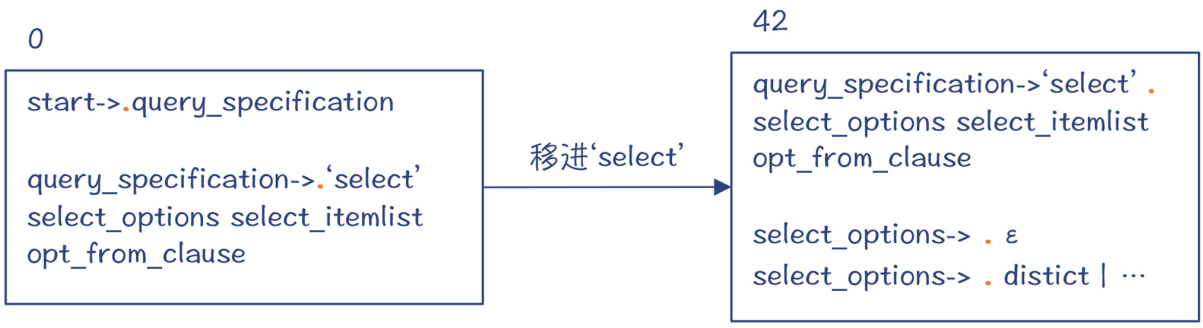


图 7：移进 select 后的 DFA

第 3 步，因为预读到的 Token 是一个数字（NUM），这说明 select_options 产生式一定生成了一个ε，因为 NUM 是在 select_options 的 Follow 集合中。

这就是 LALR 算法的特点，它不仅会依据预读的信息来做判断，还要依据 Follow 集合中的元素。所以编译器做了一个规约，也就是让 select_options 为空。

也就是，编译器依据“select_options->ε”做了一次规约，并进入了新的状态 920。**注意**，状态 42 和 920 从 DFA 的角度来看，它们是同一个大状态。而 DFA 中包含了多个小状态，分别代表了不同的规约情况。

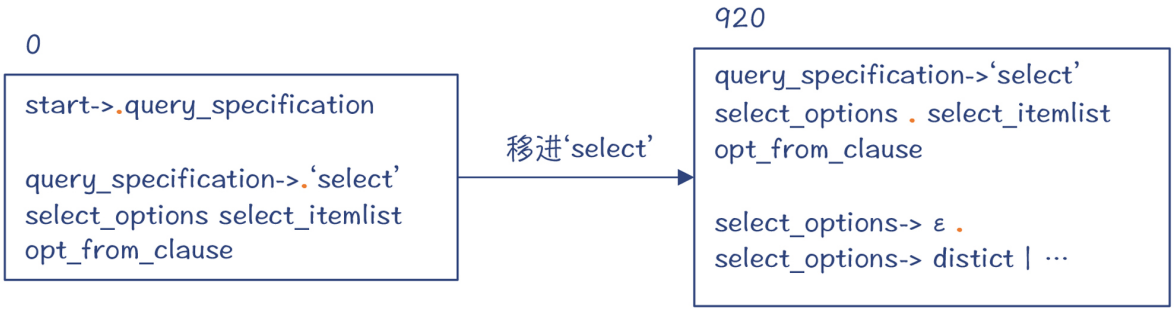


图 8：基于“select_options->ε”规约后的 DFA

你还需要注意，这个时候，老的状态都被压到了栈里，所以栈里会有 0 和 42 两个状态。栈里的这些状态，其实记录了推导的过程，让我们知道下一步要怎样继续去做推导。

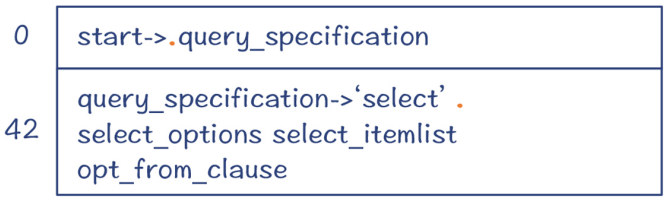


图 9：做完前 3 步之后，栈里的情况

第 4 步，移进 NUM。这时又进入一个新状态 720。

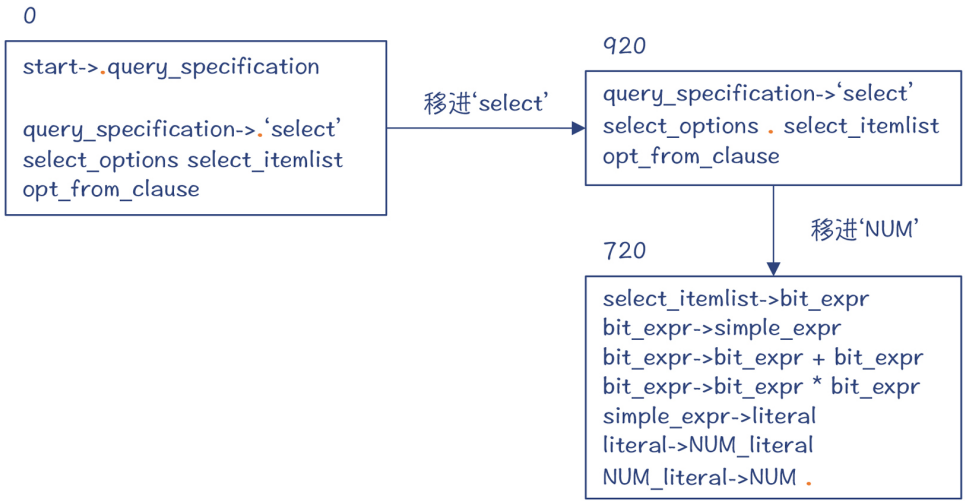


图 10：移进 NUM 后的 DFA

而旧的状态也会入栈，记录下推导路径：

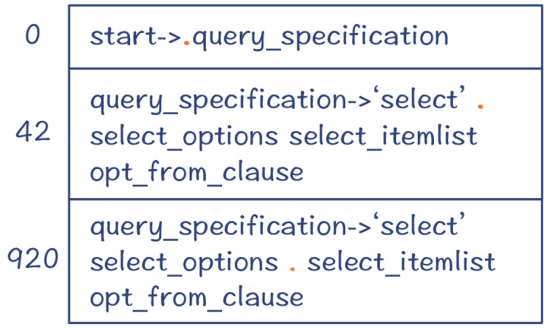


图 11：移进 NUM 后栈的状态

第 5~8 步，依次依据 NUM_literal->NUM、literal->NUM_literal、simple_expr->literal、bit_expr->simple_expr 这四条产生式做规约。这时候，编译器预读的 Token 是 + 号，所以你会看到，图中的红点停在 + 号前。

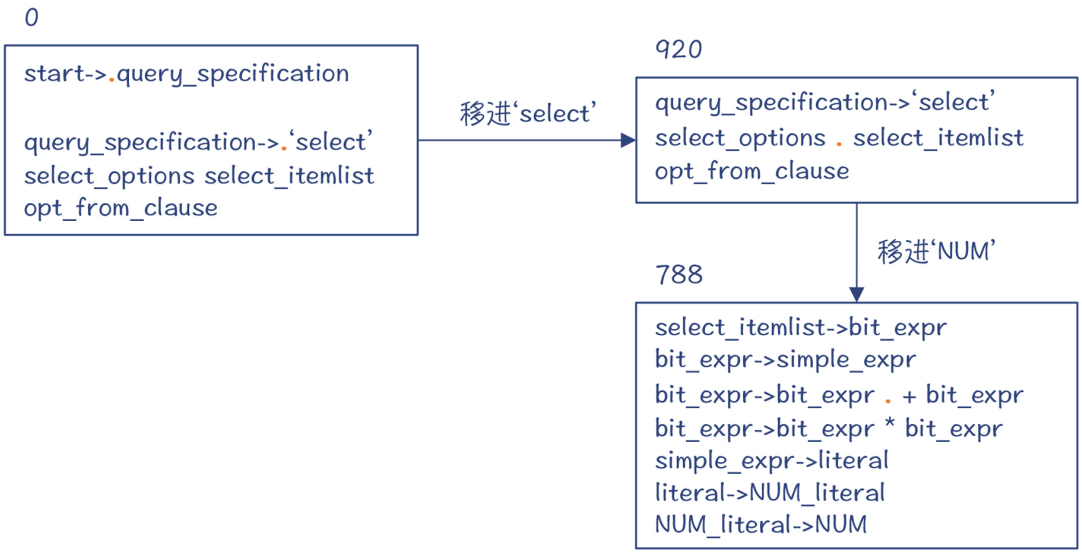


图 12：第 8 步之后的 DFA

第 9~10 步，移进 + 号和 NUM。这个时候，状态又重新回到了 720。这跟第 4 步进入的状态是一样的。

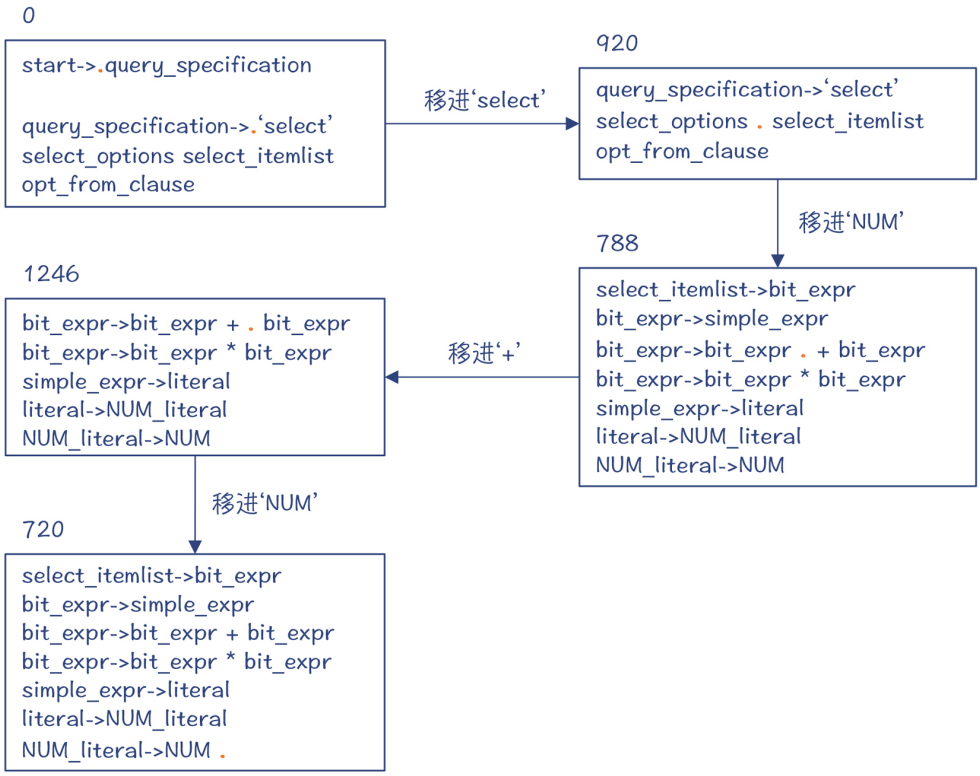


图 13：第 10 步之后的 DFA

而栈里的目前有 5 个状态，记录了完整的推导路径。

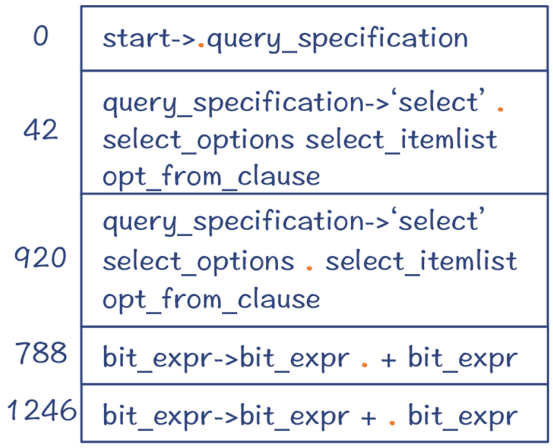


图 14：第 10 步之后栈的状态

到这里，其实你就已经了解了 LR 算法做移进和规约的思路了。不过你还可以继续往下研究。由于栈里保留了完整的推导路径，因此 MySQL 编译器最后会依次规约回来，把栈里的元素清空，并且形成一棵完整的 AST。

课程小结

这一讲，我带你初步探索了 MySQL 编译 SQL 语句的过程。你需要记住几个关键点：

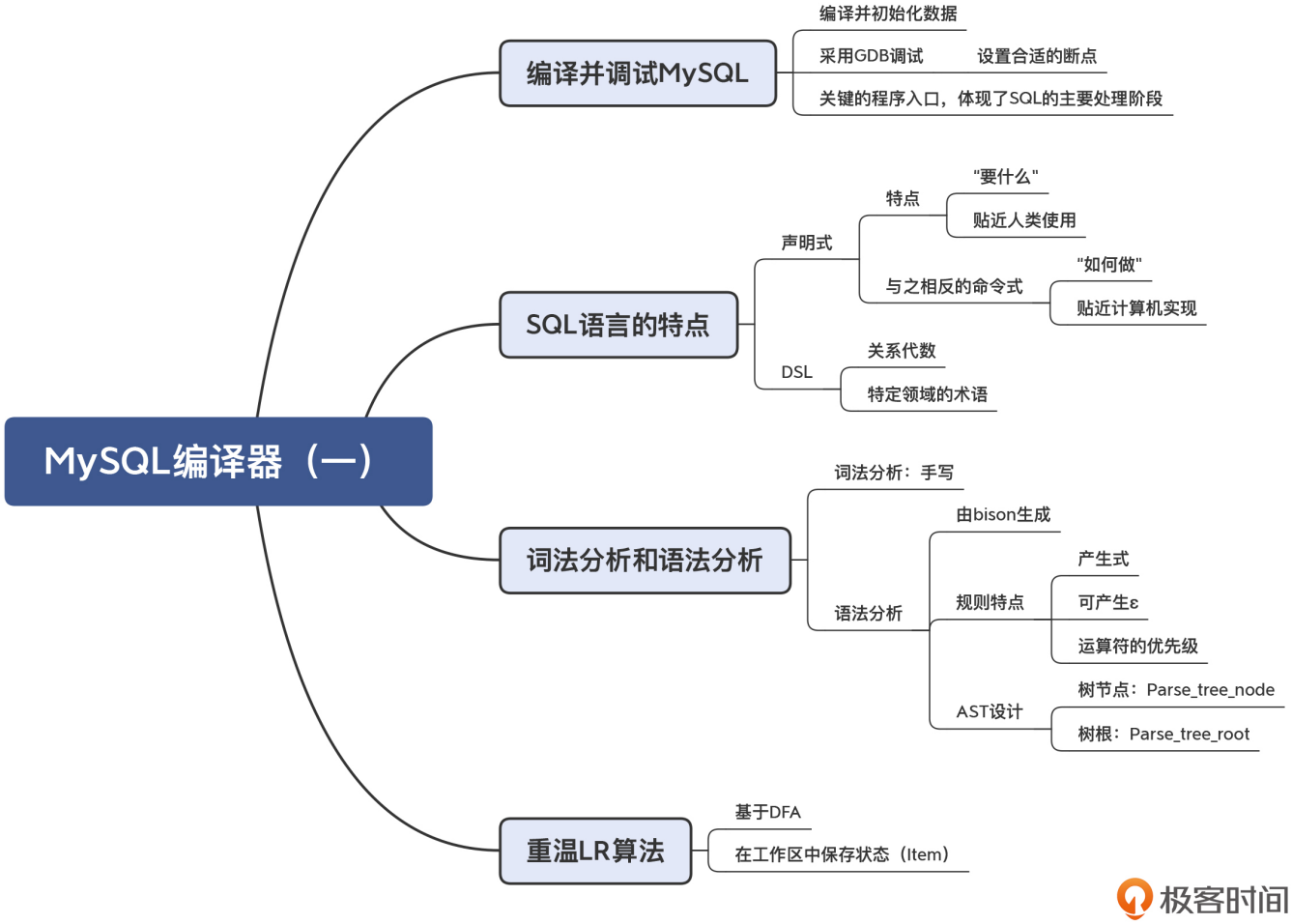
掌握如何用 GDB 来跟踪 MySQL 的执行的方法。你要特别注意的是，我给你梳理的那些关键的程序入口，它是你理解 MySQL 运行过程的地图。

SQL 语言是**面向关系数据库的一种 DSL，它是声明式的**，并采用了领域特定的模型和术语，可以为你设计自己的 DSL 提供启发。

MySQL 的语法分析器是采用 bison 工具生成的。这至少说明，语法分析器生成工具是很有用的，连正式的数据库系统都在使用它，所以你也可以大胆地使用它，来提高你的工作效率。我在最后的参考资料中给出了 bison 的手册，希望你能自己阅读一下，做一些简单的练习，掌握 bison 这个工具。

最后，**你一定要知道 LR 算法的运行原理**，知其所以然，这也会更加有助于你理解和用好工具。

我依然把本讲的内容给你整理成了一张知识地图，供你参考和复习回顾：



一课一思

我们今天讲到了 DSL 的概念。你能分享一下你的工作领域中的 DSL 吗？它们是否也是采用声明式的？里面用到了哪些特定领域的术语？欢迎在留言区分享。

感谢你的阅读。如果有收获，欢迎你把今天的内容分享给更多的朋友。

参考资料

1. MySQL 的内行手册（MySQL Internals Manual）能提供一些重要的信息。但我发现文档内容经常跟源代码的版本不同步，比如介绍源代码的目录结构的信息就过时了，你要注意这点。
2. bison 的手册。

提建议

更多课程推荐

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 24 | Go语言编译器：把它当作教科书吧

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。